AD-A268 395

WL-TR-93-1070

A SURVEY OF THREE-DIMENSIONAL PATH PLANNING IN THE CONTEXT OF A POINT-LIKE AIRCRAFT AVOIDING DANGER REGIONS

Jeffrey Alan Goldman

DTIC
ELECTE
AUG 24 1993
S B D

May 1993

Final Report for Period August 1991- January 1993

Approved for Public Release; Distribution is unlimited

AVIONICS DIRECTORATE
WRIGHT LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OHIO 45433-7409

93 8 23 130

93-19626

NOTICE

When Government drawings, specifications or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility nor any obligation whatsoever. The fact that the government may have formulated, or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication or otherwise in any manner construed, as licensing the holder or any other person or corporation, or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

This report is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.

JEFFREY ALAN GOLDMAN
Computer Scientist
Artificial Intelligence
Technology Section

JERRY L. COVERT
Chief, Information Processing
Technology Branch

CHARLES H. KRUEGER, Chief
System Avionics Division

If your address has changed, if you wish to be removed from our mailing list or if the addressee is no longer employed by your organization, please notify WL/AAAT-3, 2185 Avionics Circle, Bldg 635, Wright-Patterson AFB, OH 45433-7301 to help us maintain a current mailing list.

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	May 1993	Final Aug 91 - Jan 93

4. TITLE AND SUBTITLE
A SURVEY OF THREE-DIMENSIONAL PATH PLANNING IN THE CONTEXT OF A POINT-LIKE AIRCRAFT AVOIDING DANGER REGIONS

5. FUNDING NUMBERS
PR-2003
TA-05
WU-54

6. AUTHOR(S)
Jeffrey Alan Goldman

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Avionics Directorate, Wright Laboratory
Air Force Materiel Command
Wright-Patterson AFB OH 45433-7409

8. PERFORMING ORGANIZATION REPORT NUMBER
WL-TR-93-1070

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Avionics Directorate
Wright Laboratory
Air Force Material Command
Wright-Patterson AFB OH 45433-7409

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
WL-TR-93-1070

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Path Planning has been a topic of research in many areas including robotics and navigation. The purpose of this report is to explore the problems of three-dimensional path planning in context of a point-like airplane traveling to avoid circular danger regions. We will explore two distinct problems. The first problem is how to plan a path when the locations of all the dangers are known. The solution to this problem gives the plane an optimal path to follow before it even leaves the ground. We will refer to this as the global path planning case. In the second problem, the locations of the dangers are not known in advance. Instead, they are known to the plane when they are within a sensor range. The plane changes its path when it senses the danger areas. We will refer to the second problem as the dynamic path planning case. Both of these cases will be subject to turning constraints. The global path planning case can be solved with Collins decomposition. The dynamic path planning case, however, is still open ended. This report outlines several approaches and their pitfalls concluding with subgoal avoidance as a solution for particular classes of reconnaissance scenarios.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Path Planning, Algorithm, Potential Fields, Heuristic, Digital Terrain Elevation Map, Collins Decomposition, Subgoal Avoidance.	65
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18 SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Accession For
NTIS GRA&I
DTIC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or Special
A-1

DTIC QUALITY INSPECTED 3

Contents

Chapter

1. Introduction **1**
1.1. Overview 1
1.2. Motivation 2
1.3. Discussion 2
1.4. Program Overview 2
1.4.1. The System 2
1.4.2. The Software 3

2. Visiting Goals **4**
2.1. The Traveling Salesman 4
2.2. The Greedy Alternative 4

3. Turning Constraints **6**
3.1. Overview 6
3.2. The Constraint Algorithm 6

4. Finding a Path with Foreknowledge of Dangers **8**
4.1. Overview 8
4.2. The Methods Without Any Dangers 8
4.2.1. Midpoint Line Approach 8
4.2.2. Straight Line Approach 9
4.2.3. Greedy Coordinate Line Approach 10
4.3. The Methods to Avoid Dangers 11
4.3.1. Potential Fields 11
4.3.1.1. Infinite Cylinder *12*
4.3.1.2. Infinite Pillar *12*
4.3.2. Collins Decomposition 13
4.4. Summary 13

5. Finding a Path with Restricted Foreknowledge of Dangers **15**
5.1. Overview 15
5.2. Collins Decomposition 15
5.3. Potential Fields 16

5.4. Subgoal Avoidance 16
5.4.1. Overview 16
5.4.2. The Algorithm 16
5.4.3. The Problems 18

6. Conclusions 20
6.1. Finding a Path with Foreknowledge 20
6.2. Finding a Path with Restricted Foreknowledge 20
6.3. Future Work 21

7. References 23

Appendix

A. The User Menu 24

B. Simulations 26

C. The Software 29
C.1. Constants.h 29
C.2. Coordinate.h 30
C.3. Global.h 32
C.4. Line.h 36
C.5. List.h 38
C.6. MidpointLine.h 39
C.7. NewLine.h 40
C.8. Planner.h 43
C.9. Plotter.h 46
C.10. Potential.h 47
C.11. Reader.h 49
C.12. Turnaround.h 50
C.13. User.h 52
C.14. Y_Equal_X.h 55

List of Figures

5.1. Path Trap **17**

5.2. Subgoal Avoidance **18**

B.1. Simulation 1 ..... **27**

B.2. Simulation 2 ..... **28**

Chapter 1

Introduction

1.1. Overview

We wish to explore in detail some of the problems of three-dimensional path planning. In order to do so, we must define the problems precisely. For the purpose of this report, we will think of path planning in the context of a starting location, an ending location, a list of locations to visit, and a list of locations to avoid. There are two main distinctions that we will explore in three-dimensional path planning. They are: 1) constructing a path where we know everything about our environment in advance (the global path planning case), and 2) constructing a path where we know everything about our environment except we only know about the list of locations to avoid within a certain radius of our current position (the dynamic path planning case). In addition, we will impose turning constraints in both cases. The scenario we will explore in this report is the problem of an airplane traveling on a reconnaissance mission.

The methods that solve the dynamic path planning case may also be relevant to robot motion planning in the two-dimensional case. This case may also be applicable to navigational problems in commercial aircraft where the dangers can be thought of as unexpected storm fronts. When exploring how to construct a path that visits several places, the question of optimality comes into play. We will not discuss any solutions to path planning in the context of the *Traveling Salesman Problem* [1], but instead focus on a *greedy heuristic* for the next goal to visit.

1.2. Motivation

This report is relevant in the following context. Given terrain, a list of goals, and a list of dangers for our plane, problems arise that we would like to know how to solve. First, in the global path planning case, how do we construct an efficient plan that the pilot of our plane can follow to visit the goals and avoid the dangers? Second, in the dynamic path planning case, how do we alter our plan on the fly so that the pilot can safely avoid the dangers? We will be exploring these two issues by creating a simulation program of a plane that flies over a mountainous terrain.

1.3. Discussion

It turns out that in the case where everything is known in advance, the problem has already been solved. In the dynamic path planning case, when turning constraints are imposed and an alternative path must be created on the fly, the problem is still open ended. If the turning constraints on the second case are relaxed, we can solve the problem. However, we are looking at an airplane pilot flying a reconnaissance mission so turning constraints must be met. Also, the plan must be updated in real time because of the implicit constraint of pilot safety.

It is relevant to follow the progression of problems that arise starting with the simpler case first. Although the global path planning case has already been solved, there is still some merit in a discussion of that particular scenario. We will then discuss and analyze a solution to the second problem. The solution is not an all encompassing one, but it is effective in a particular class of reconnaissance scenarios.

1.4. Program Overview

1.4.1. The System

The system we will be using to simulate a plane flying over a terrain is the Silicon Graphics Indigo (IRIS) [2]. This is an IRIS graphics machine capable of 24-bit color. The IRIS was chosen for its excellent graphics capabilities. Visualizing path planning is much easier when you can see precisely what is going on. The computer language will be AT&T Unix System Laboratories C++ release 3.0 [3] [4]. This language was chosen for its modular ability; classes

and subclasses are easily added. This language also lends itself well to the Silicon Graphics Package.

The environment is a digital terrain elevation map taken from the US. Geological Survey[5]. Specifically, it is a map of Greeley, Colorado. It is a grid bounded by 104 degrees to 105 degrees longitude and 40 degrees to 41 degrees latitude. The map is approximately 54.25 miles by 67.5 miles. There are 1201 x 1201 elevation points to represent this grid. The scale is roughly 1 mile = 200 pixels. For our purposes, we may consider the map to be of sufficient size for a typical reconnaissance mission. The actual elevation data were taken to get an approximate idea of typical terrain.

It seemed easier to take actual data than to try to generate synthetic data. Greeley, Colorado, was chosen because it is a mountainous area of the United States. This is useful if we wish to explore scenarios where we would like to impose an altitude constraint.

1.4.2. The Software

The software I wrote for this simulation is discussed in detail at the end of the paper. At this point, only a brief summary is necessary. The program is interactive with the user. One can choose a method of path planning as well as the locations to visit and avoid. After the navigation methods are chosen, the user can plot the terrain complete with goal locations, danger locations, and paths planned for a full visual representation.

The software is modular in design complete with a coordinate class to represent a point object. This class contains methods to calculate distances to other points for convenience. There is also a planner class to incorporate all of the classes that solve the subproblem of traveling from one point to another. A turnaround class is also provided to handle the strict turning constraints.

Chapter 2

Visiting Goals

2.1. The Traveling Salesman

The problem of being given a starting point, an ending point, and a list of goals to visit in an optimal way, is precisely the Traveling Salesman Problem with equal cost nodes. Since we already know that this problem is NP-complete [1] and that we ultimately want real-time answers to our problem, we must use some other technique than a brute force method for finding a path. In the global path planning case, we can use a brute force method to find an optimal path because time is not a constraint. In fact, we could use a brute force method in the other case as well to give the pilot an initial plan that is optimal. However, in this report, finding optimal paths are less of a concern than avoiding dangers. Therefore, we will use a greedy heuristic instead, knowing that we can always start with an optimal path from some brute force method if desired.

2.2. The Greedy Alternative

The greedy heuristic works as follows: choose a goal from the goal list that is the closest to our current position; delete this goal from the list; when we arrive at that goal, we repeat this procedure. One caveat to this method is that the distance formula the greedy heuristic uses calculates a two-dimensional distance. Our current position and our goals are all defined by a point in three-space, i.e., having x, y, and z -coordinates. The definition of distance in three-space is:

$$\text{Distance} = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2 + (z_1 - z_0)^2}$$

The definition of distance in two-space is:

$$\text{Distance} = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}$$

The reason we will use the distance in two-space is because we should not visit a goal that is far away at the same altitude before we visit a goal that is very close, but on top of a very high mountain peak. Using a three-dimensional distance would be less efficient than a two-dimensional distance. In a mountainous terrain, altitude values might be 4 times the magnitude of the *(x,y)*-coordinates and thus dominate the distance formula. In a relatively flat terrain, the altitude would be insignificant, so the two-dimensional formula and the three-dimensional formula are essentially the same. Therefore, "closest" in the greedy heuristic, will use two-dimensional distance as a metric.

Another caveat to this heuristic is that the distance metric in the actual program will find the square of the two-dimensional distance. This is because computing the square root of a number is expensive; it would slow down the algorithm. Finding the closest goal to one's current position using the square of the two-dimensional distance as a metric will be our approach throughout the report when concerning traveling from goal to goal. One can also see the catastrophic effects of using the three-dimensional distance when we do not compute the square root.

Chapter 3

Turning Constraints

3.1. Overview

The turning constraints will be simplified from real aircraft turning ability so that we can study methods for planning with constraints in general. We will restrict our plane to a 45 degree turn with a minimal traveling distance of 30 pixels. This traveling distance is the minimum distance that the plane must travel before it can turn 45 degrees. In other words, if the plane has to make a 90 degree turn, the plane is forced to travel as follows: 1) the plane must travel in its current direction for 30 pixels, 2) the plane must travel 45 degrees from the normal for 30 pixels, 3) the plane must travel 45 degrees from its previous position for 30 pixels. In a sense, we have a restricted regular octagon of travel where each side is length 30.

The reason we impose a constraint in this way is that it makes things easier to graph. With these turning constraints, we do not have to worry about plotting curve surfaces or how to generate them. Since the minimal traveling distance is adjustable by the user, this is a reasonable simplified model of what real plane turning constraints are like.

3.2. The Constraint Algorithm

To set up the turning algorithm, we created a class called *turnaround*. The method that does the actual turning takes three points. The first point is a point that was already in the planned path. We try to make this point about 10 points prior to the current position since this point will

be used in determining the direction we were currently traveling. The next point is the current position. The last point is the location of the place we wish to go. The turning algorithm works as follows:

1) Use the first and second point to determine the current direction of travel. We accomplish this by finding the line in two-space between the two points ignoring the altitude. Using the slope, m, we can figure out the current direction of travel as follows:

$m \leq -2$ or $m \geq 2$	North or South
$-0.5 \leq m \leq 0.5$	East or West
$0.5 < m < 2.0$	Northeast or Southwest
$-2.0 < m < -0.5$	Northwest or Southeast

We then look at the differences in the x and y-coordinates of the two points to choose between the two directions. For example, in the North or South case, if the y-value is increasing, we choose North. Using the slope and the differences, we can find the current direction of travel.

2) We then use the same method described above on the second and third points to find the desired direction of travel.

3) Once we know which way we were going, and which way we wish to go, we calculate the turning direction (clockwise or counterclockwise) that yields the least number of turns and that is the direction we will go.

4) When the algorithm is finished, the turning positions are added on to the plan and the current position is returned.

Chapter 4

Finding a Path with Foreknowledge of Dangers

4.1. Overview

Since the greedy heuristic takes care of choosing the goals to visit, and the turnaround class takes care of our turning constraint, the problem reduces to the local problem of finding a path from one point to another. Here, we will explore some basic methods of traveling from one point to another in the case where there are no dangers and we will then discuss how to go about modifying these methods in order to avoid dangers.

4.2. The Methods Without Any Dangers

4.2.1. Midpoint Line Approach

This approach is taken from a typical graphics algorithm whose purpose is to draw a straight line. This method is sometimes called *Scan Line conversion* [6]. The algorithm is presented here:

```
dx = x_destination - x_current
dy = y_destination - y_current
d = 2*dy - dx
incrE = 2*dy
incrNE = 2*(dy-dx)

while ( x_current < x_destination )
{
     if ( d<=0 )
     {
          d=d+incrE
          x_current=x_current+1
```

```
        }
        else
        {
            d=d+incrNE
            x_current=x_current+1
            y_current=y_current+1
        }
    }
```

Unfortunately, this algorithm will not work for all possible endpoints. It will only solve the problem of drawing a line in the East or Northeast direction. From this point on, we will never again use this algorithm. However, this was the first step in attempting to solve this problem.

4.2.2. Straight Line Approach

This next approach was an adaptation of the Scan Line Conversion algorithm. The *Straight Line Approach* is almost exactly how it sounds. The only caveat is that this approach will construct a straight line in two dimensions. If we wish to travel from (x_0, y_0, z_0) to (x_1, y_1, z_1), we simply find the equation of the line in the x,y-plane and add points to our path along that line. The altitudes are set to be the actual altitudes of the terrain.

It is clear that by taking the altitudes of the terrain, this model appears to be one that is more like a car driving over hills and mountains than a plane. For now, we will not be concerned with the altitude that the plane flies. We can assume that we always fly a certain height above the ground. In fact, if the constraint is added that the plane must fly no higher than a specific value, we will be faced with a problem more difficult than avoiding dangers.

The Straight Line Approach looks at the value of the slope and depending on its value, we may increment or decrement y from its previous value and use the equation to find x, or vice-versa. For example, if the slope of the line is 10 and we are traveling in the positive y-direction, it makes more sense to increment y by one and find x than it does to increment x and find y. This way, we get a more accurate representation of this straight line and we fill in the path with more points that are true to that line.

The algorithm is shown here:

```
while    current_x!=destination_x or
         current_y!=destination_y
{

    dx = current_x - destination_x
```

```
        dy = current_y - destination_y

        if horizontal line
             dx < 0 => current_x=current_x+1
             dx ≥ 0 => current_x=current_x-1

        if vertical line
             dy<0 => current_y=current_y+1
             dy>=0 => current_y=current_y-1

        All other cases:
        -1 <= m <= 1
             dx<0 => current_x=current_x+1
             dx>=0 => current_x=current_x-1

             y = round (mx + b)
        else
             dy<0 => current_y=current_y+1
             dy>=0 => current_y=current_y-1

             x = round ( (y-b)/m )
}
```

Given a starting point, an ending point, and a list of goals to visit, we now have a way to construct a path that will start at our starting point, end at our ending point, and visit all of the goals while constrained by the turning constraints. We can use the greedy heuristic for goal selection, the Straight Line Approach for finding a path from point to point, and the turning constraint to ensure that our path does not do anything our fictitious plane is not allowed to do. Our path is not by any means an optimal one in terms of the Traveling Salesman Problem nor is it optimal in terms of the plane having to travel up and down steep hills. In fact, at this point, we cannot even avoid a single danger point. However, this is our foundation for more effective path planning.

4.2.3. Greedy Coordinate Line Approach

Before we look at methods of avoiding danger points, there is one other method that this researcher used to solve the subproblem of traveling from point to point. Although it does not perform as well as the Straight Line Approach, it might find a path more quickly. This method is called the *Greedy Coordinate Line Approach*. It takes a starting point, (x_0, y_0), and an ending point, (x_1, y_1). The method then increments or decrements x_0 and y_0 by one, depending on x_1 and y_1, until the coordinates match. This amounts to traveling on the line $y = x$ or $y = -x$ until one of the coordinates matches x_1 or y_1, and then traveling in a horizontal or vertical line the rest of the

way. This method handles the altitude in the same manner as the Straight Line Approach, copying altitudes from the terrain. This path is subject to the same changes in altitude as well.

The advantage to this method is that the algorithm does not have to find the equation of a line. It only has to know its current position and its goal position coordinates. The disadvantage is of course, the path is always less optimal than the Straight Line Approach except in the case when the starting point and goal point are on the line $y=\pm x$.

The algorithm is presented here:

```
while dx!=0 or dy!=0
{
     dx = current_x - destination_x
     dy = current_y - destination_y

     if (dx < 0)
          current_x=current_x+1
     else
          if (dx > 0)
               current_x=current_x-1

     if (dy < 0)
          current_y=current_y+1
     else
          if (dy > 0)
               current_y=current_y+1
}
```

We now have two methods to compare to each other in the simple case of traveling from goal to goal without any dangers. The next section applies to both methods; however, our focus will be on the Straight Line Approach.

4.3. The Methods to Avoid Dangers

We now have two ways to travel where we start somewhere, end somewhere, and visit a list of goal sites. The question now is how do we also avoid danger sites? The methods explained here are the methods that this researcher explored and extrapolated.

4.3.1. Potential Fields

The first major technique explored was the notion of setting up potential fields around the danger points ahead of time, before any path planning was done. The idea is to have a two-dimensional array with each dimension representing x and y-coordinates. The array value is the

potential field value at that point. Points that are close to the danger point are given high values and all other points are given values of zero. Then, while the path is being made, if the potential field value at that point is higher than a threshold value, the path is altered [7].

For now, in the case where we use potential fields to avoid danger points, we will pretend that there are no turning constraints on the plane. Ultimately, it will turn out that potential field methods will break down when turning constraints are added. Nonetheless, potential fields are a good way to avoid danger areas in general. They will provide a useful insight to methods that will be able to handle the turning constraints.

4.3.1.1. Infinite Cylinder

The infinite cylinder was set up as follows. Any point that was within some fixed two-dimensional radius of a danger point was set to a high value. All other points were set to zero. With the turning constraints turned off, if the path ran into one of these high points, a point was selected in one of the eight directions (N, S, E, W, NE, NW, SE, SW) as an alternative. The path then tried to continue along this new line formed by this new point and its original goal point in the case of the Straight Line Approach. In the case of the Greedy Coordinate Line Approach, it simply continued from this new point. To prevent backtracking, after each point was added to the path, its potential field value was changed from zero to the high value.

The field created is like an infinite cylinder centered about the danger point. Nothing is allowed to pass through it. Everything that would have gone through it, must now go around. It is important to note that by preventing backtracking, we are setting up infinite walls that can also not be traveled through.

This approach is not the best way to implement this procedure. A better way to do this is to use a steepest gradient approach [7]. For this application, planning a path when everything is known in advance, traveling along the steepest gradient is the approach to use. The reason the above-mentioned procedure was used instead, is because for the dynamic path planning case, we need an on-the-fly method to handle it.

4.3.1.2. *Infinite Pillar*

This potential field is similar to the above except we make a square around the danger in the x, y-plane and extend it infinitely in the z-directions. The only reason it is mentioned here is that it is easier to compute than the aforementioned cylinder. In practice, any topology can be used to block off regions that we wish to avoid. Given ample time to plan the path in the global path planning case, we can use the *steepest gradient approach* [7] to find a path that suits our needs. In fact, we can use this idea to restrict the height that the plane is allowed to fly. We simply create regions around the peaks of our mountains. These regions might be irregular, but without the turning constraints, we can find a path that has an altitude restriction.

4.3.2. Collins Decomposition

It turns out that using potential fields or traveling along steepest gradients is not the best way to solve the problem of finding a path in the global path planning case. The potential field method cannot handle the turning constraints unless the fields are tailored to force them. It may be necessary to set up complicated regions with field values other than just a high value and a zero value. A better way to do this is to use *Collins decomposition* [8].

This method looks at the entire terrain. It then breaks down the terrain into regions that are safe to travel in and into regions that are not safe to travel in. The algorithm then tries to piece together the safe regions and create a path by connecting curves in each of those regions. It is possible to impose turning constraints and it is also possible to restrict the plane to some altitude. For example, we may want the plane to complete its mission by flying as low to the ground as possible to avoid radar. This can be accomplished by setting up unsafe regions that are above some altitude.

Ultimately, if we are in the global path planning case, the plane has a limited turning radius, and we wish the plane to fly as low as possible, Collins decomposition with some modifications will solve the problem.

4.4. Summary

In the case of finding a path with foreknowledge of dangers, we can use potential fields or Collins decomposition. With potential fields, we talked about setting up infinite regions and

alluded to a steepest gradient approach. Since it turns out that Collins decomposition will solve the problem including turning constraints, we need look no further. The real question now is how to adapt the aforementioned methods to the problem of finding a path in the dynamic path planning case. For this problem, we must adjust the path on the fly.

Chapter 5

Finding a Path with Restricted Foreknowledge of Dangers

5.1. Overview

The problem will be set up as follows. The airplane will know about the terrain, where the starting point is, where the ending point is, where the list of goals are, and the plane will be restricted to turning constraints just as the other scenario. However, the plane will only know about a danger point if it is within some fixed two-dimensional radius of its current position. As soon as it knows, it can alter its course to try to avoid the danger. Here, we will examine the methods of avoiding dangers in the previous scenario and see how they apply. We will also look at a new method that this researcher suggests as a way to handle some classes of scenarios.

5.2. Collins Decomposition

Collins decomposition does not solve the problem in the dynamic path planning case because it is a double exponential time algorithm in the number of regions that divides up the terrain [8]. When everything is known in advance, we have all the time in the world to make our plan. However, in this case, we need to alter paths on the fly. In this scenario, the algorithm must run in real-time. We must turn to another method to solve our problem.

5.3. Potential Fields

Using the steepest gradient approach, we have the problem of how to handle the turning constraints. Originally, this algorithm was intended for a robot arm that had no such constraints. It is not clear how to modify this algorithm in order to accommodate the turnaround class. Also, this method would not be able to run in real time. It is an exponential time algorithm in the number of vertices of the danger region.

The method that creates the path on the fly has problems of its own. If we temporarily ignore turning constraints, we run into problems when danger regions overlap. In the case of the infinite cylinder, when two dangers overlap, a concave danger region is created. The on-the-fly potential method of avoidance can end up trapping itself in this region (See Figure 5.1). There are also possible one way doors that could trap the plane.

If we ignore the overlapping dangers case, and try to solve the turning constraint problem, we need to set up a potential field of two concentric infinite cylinders so that when the plane enters the outer cylinder, it can turn to avoid the danger. Unfortunately, we have to set up these potential fields in real time which is likely to be extremely difficult. It may be possible to set up concentric pillars; however, we may introduce similar regions as described above.

5.4. Subgoal Avoidance

5.4.1. Overview

Instead of creating potential fields to avoid oncoming dangers, why not simply find some intermediate point that avoids that danger? Then, when the plane reaches that point, it can continue from there to its original goal. This is the basic outline of *subgoal avoidance*.

5.4.2. The Algorithm

The subgoal avoidance algorithm works as follows. We use one of the methods to find a path from one point to another. Then, when the plane is at a position where it can sense a danger, we check to see if our current line of travel will enter the danger region. If it does not, we ignore it. If it does, we find a subgoal that avoids this danger. When we reach this subgoal, we use one of the methods to find a path between this new point and the original destination.

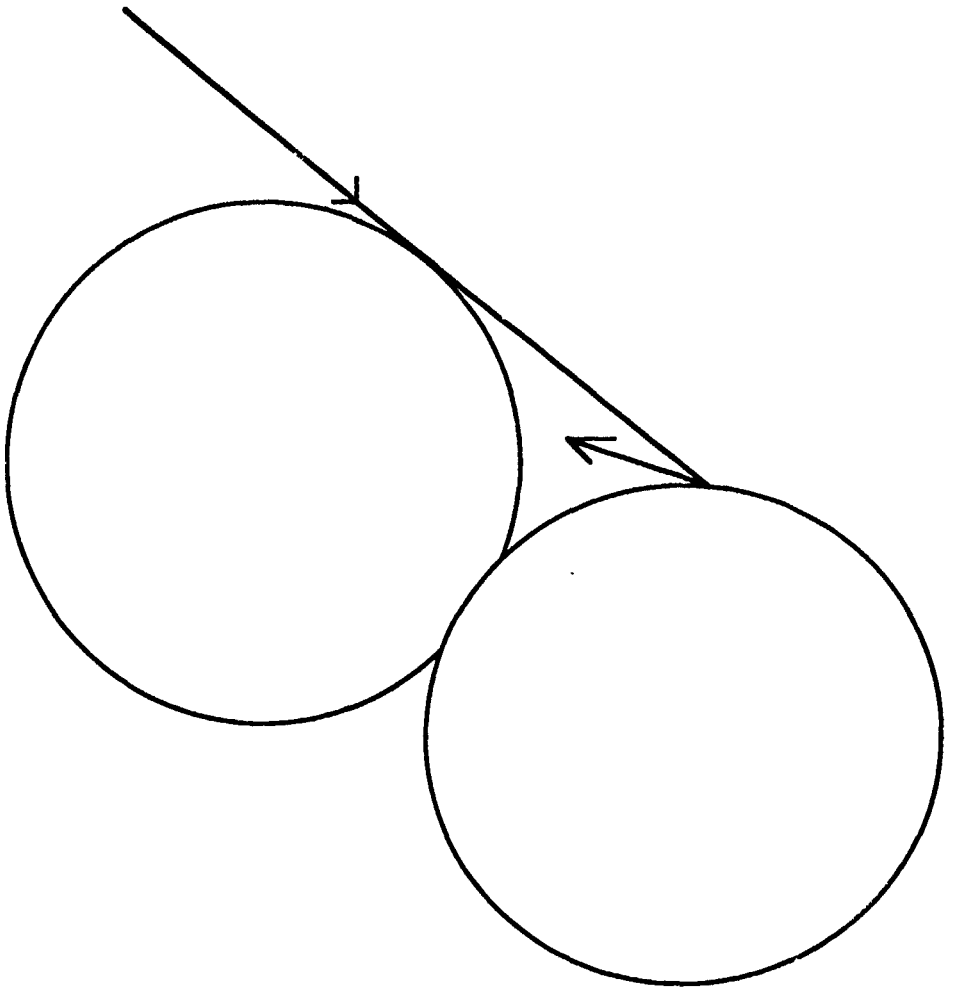

Figure 5.1: Path trap

The way we find this subgoal is as follows. We look at the line that is perpendicular to our current line of travel and intersects the danger point. We then compute the distance, d, of the danger point to our current line of travel. The subgoal lies on the perpendicular line at some multiple of d (See Figure 5.2). This is how we find a subgoal. Note that the lines and distances we are talking about lie in the two-dimensional projection of this three-dimensional problem.

It is relevant to point out that actual distances are not computed because the square root operation is too expensive. In actuality, we compute the square of the distance. Also, the distance, d, is figured out by symmetry. This is also cost efficient.

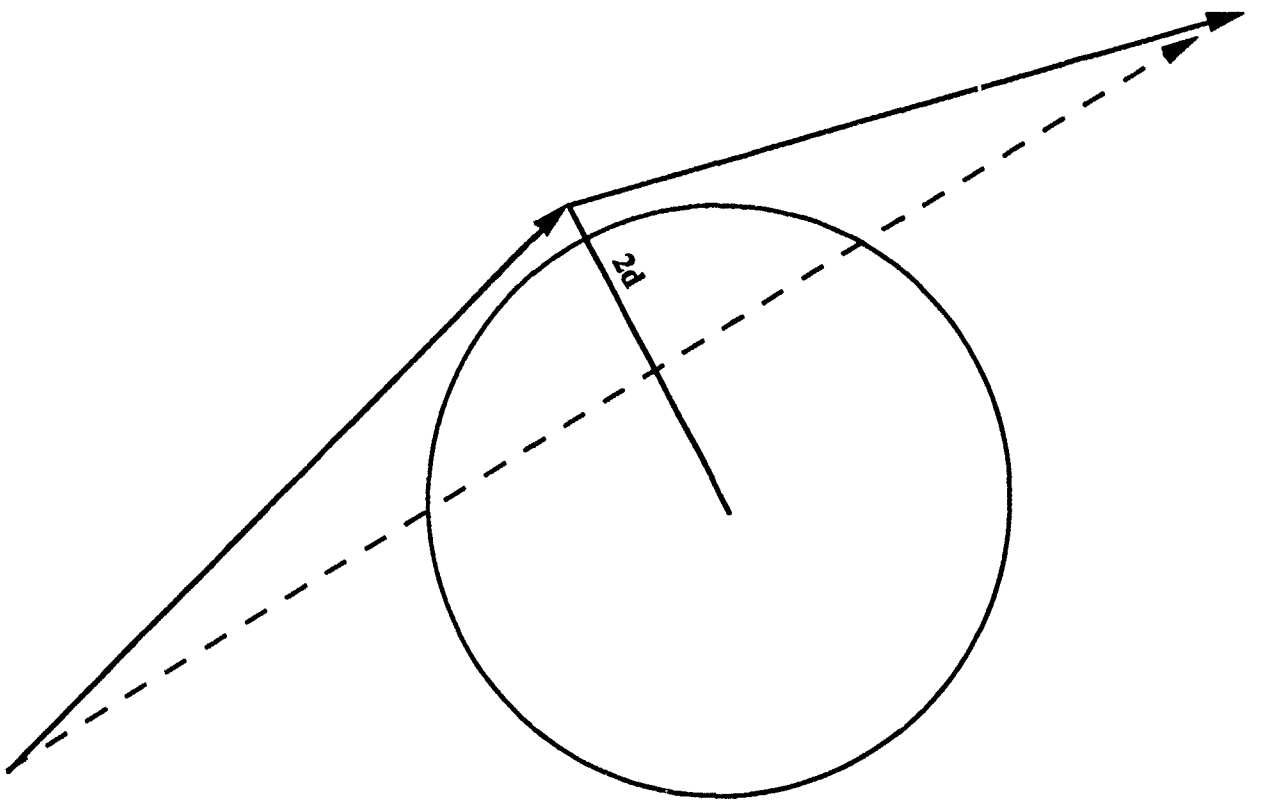

Figure 5.2: *Subgoal avoidance*

5.4.3. The Problems

This method does not handle all problems with potential danger hazards. First, if we look at the case when a subgoal is created in the path of another danger point, we may incur problems. The algorithm may realize that another danger is in its path and try to correct it. However, the new subgoal may send the plane back to the original danger. This problem arises when two or more dangers are close together or overlapping.

Another problem may occur. If the subgoal position is close to a danger, but not in the danger area, the plane may reach that subgoal and find that it cannot avoid this new danger while keeping to its turning constraints. This problem will occur when dangers are close and positioned in such a way that causes the algorithm to fail. This problem is essentially the problem of starting (or ending) the plane too close to a danger region or positioning a goal too close to a danger position.

We may also have a combination of both problems mentioned above. We may have overlapping dangers where we create a subgoal that sends us in front of yet another danger point.

The real problem here is that there is no guarantee that this algorithm will work for all possible solvable scenarios. It is clear that if we have a goal to visit that lies in the radius of a danger area, we cannot visit that goal and thus we cannot solve that scenario (unless, of course, we take into account those types of situations). It is also possible to position several dangers to create an overall danger area that is designed to lure in the plane only to trap it. The upshot is that we can only guarantee success if the set of points that includes the starting point, the ending point, the dangers, and the goals, is sparse.

If we examine this condition in more detail, we can be sure of success as long as each object is twice the diameter of the plane turning radius away from any other object. We also need a reasonable constraint on plane sensor ability. We would like the sensors to be able to detect dangers before we are within a distance equal to twice the diameter of the plane turning radius from the danger. This way, the plane has enough room to turn after it reaches a goal or a subgoal.

Chapter 6

Conclusions

6.1. Finding a Path with Foreknowledge

In the global path planning case where the entire scenario is known in advance, the problem has been solved. We can use Collins decomposition. This method, if adapted, can handle turning constraints and altitude constraints to find an optimal path. The key reason is because we have as much time as we need to plan a path.

6.2. Finding a Path with Restricted Foreknowledge

We have seen that potential fields create problems in two ways. First, it may create convex regions or one-way doors that will allow the plane to trap itself. Second, it may take too much time to create these fields as soon as a danger is sensed. This would make it too difficult to come up with a revised plan on the fly. Collins decomposition is an exponential time algorithm that also poses the same problem as the potential fields.

For the dynamic path planning case, our fastest solution is the subgoal avoidance mechanism. We are assured a safe, on-the-fly path as long as we have a reasonable detection radius and the dangers and goals are far enough away from each other so as not to cause problems. This may appear to be a poor solution; however, for our purposes, these assumptions are reasonable. When we have a plane flying a reconnaissance mission that does not know where all of the danger zones are, it is reasonable to assume that the plane can take rather sharp turns if necessary and

that the danger zones will not appear very close together. It may also be the case that we only wish to visit a couple of goal positions. This increases our effectiveness of the algorithm. The best solution to this problem is to use the Collins decomposition as a starting path and then if dangers appear, we can deviate from our path at this point using subgoal avoidance.

6.3. Future Work

The subgoal avoidance method has its place for three-dimensional path planning. However, this is not a method that this researcher would be willing to stake his life on as the pilot of a reconnaissance mission often does. Therefore, I propose some possible follow-ups to this discussion.

One possibility is to use the Collins decomposition path plan to solve the problem and then update the plan on the fly. Instead of abandoning the plan when we come into contact with a danger point, as mentioned previously, we can instead create some subgoal that is close to the original plan. Afterwards, we try to continue to follow our original plan. Updating plans as a methodology is used in other procedures [8]. It may be that we can still create an on-the-fly planner.

Another possibility is that a potential field-like method will work fast enough for a real time application. In an article by Jean-Claude Latombe [9], he suggests a fast path planner for a car-like robot. This robot, similar to the plane, has turning constraints. The algorithm is meant to be a planner where all obstacles are known in advance. However, Latombe concludes that the algorithm on a fast enough machine can run multiple times, re-planning on line as unexpected obstacles are brought into play. The planner is for a two-dimensional universe but it seems possible to apply this to a three-dimensional scenario. This approach appears to be the most promising.

It would also be relevant to continue work in the areas of optimal path planning with additional field constraints. This researcher has only considered a path with a single turning constraint. The problem appears to be much more difficult if we add constraints such as an

altitude constraint, a fuel consumption constraint, and a time constraint. Each constraint mentioned is very important in the context of path planning and needs to be addressed.

7. References

[1] Thomas H. Cormen, et al., *Introduction to Algorithms*. MIT Press, McGraw-Hill 1991. pp. 946-960, 329–355.

[2] Patricia McLendon. *Graphics Library Programming Guide*. Lorrie Williams Production 1991.

[3] Ira Pohl. *C++ for Pascal Programmers*. Benjamin/Cummings Publishing Company, Inc. 1991.

[4] Steven C. Dewhurst, et al., *Programming in C++*. Prentice Hall 1989.

[5] U.S. Department of the Interior, U.S. Geological Survey. *Digital Elevation Models. Map #40104, Small Scale 1:250,000. One degree by one degree units*. Earth Science Information Center, Greeley, Colorado, 1992.

[6] James D. Foley, et al., *Computer Graphics Principles and Practice*. 2nd ed., Addison-Wesley, 1990. pp. 72–81.

[7] Jen-Hui Chuang. *Path Planning using the Newtonian Potential*. University of Illinois, 1991.

[8] Jean-Claude Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, 1991.

[9] Jean-Claude Latombe. "A Fast Path Planner for a Car-Like Indoor Mobile Robot," Proceedings of the 1992 National Conference on Artificial Intelligence, Anaheim, California, pp. 659–665.

Appendix A

The User Menu

The following text is what the user would see when the program is executed. The menu is designed to be easy to follow and understand. Several options are available.

```
It is assumed that the name of the map file is digitaldata.
The first coordinate Map[0][0] is: 2746
Data was successfully read.
Enter a choice (0 to end program).

List operations.
1) Read in Starting Point.
2) Read in Ending Point.
3) Read in a new goal list.
4) Read in a new danger list.

Graphic operations.
5) Plot the Start/End/Goals/Dangers/Plan/Map.
6) Set the Grain. (Grain is currently 1 pixels)
7) Toggle Plotting Danger Region.  (Plotting is on)

Text operations.
8)  Print the StartingPoint.
9)  Print the EndingPoint.
10) Print the Goals.
11) Print the Dangers.
12) Print the planned route (Plan List).

Plane constraints
13) Set Turning Length. (Length is currently 30 pixels)
14) Set line of site or radar. (Site is currently 200 pixels)
    (This is for use with methods that know where the dangers are.)

Navigation.
15) Greedy Straight Line Approach.
16) Greedy MidPoint Line Approach. (may not find path)
17) Greedy Coordinate Line Approach.
18) Greedy Straight Line Approach with a potential field.
19) Greedy Coordinate Line Approach with a potential field.
```

```
20) Greedy Line Approach with turning around danger constraints.
    (This method knows where the dangers are.)
21) Greedy Line Approach with turning around danger constraints.
    (This method only knows where the dangers are up  to a radius.)

Choice is (0)
```

Appendix B

Simulations

The first picture (Figure B.1.) is of the Straight Line Approach (green line) and the Greedy Coordinate Line Approach (blue line). This simulation had a list of goals to visit (blue circles) and no danger areas. The plane was to start at the top left corner and to end at the bottom right corner while visiting a list of goals. The turning constraint was a length of 30 pixels. The second picture (Figure B.2.) is a simulation that had the same starting position, ending position, list of goals, and turning constraint length. However, there were also a list of danger points (yellow) and a region around them which represented their effective radius. This region had a radius of 40 pixels in all cases. All are straight line approaches. The green line used an infinite cylinder potential field without the turning constraint, the blue line used subgoal avoidance in the case where all of the dangers are presumed to be known in advance, and the purple line used subgoal avoidance, but only knew of the existence of a danger point if it was less than 200 pixels away. Both subgoal avoidance methods met the turning constraint.

Figure B.1: Simulation 1

Figure B.2: Simulation 2

Appendix C

The Software

The following sections are the actual header files taken from this researcher's program. Each header is fully commented and explanations are given as to what each class is supposed to do.

C.1. Constants.h

```
#ifndef CONSTANTS
#define CONSTANTS


//
//
// GRID_SIZE is the size of the map in pixels.  The map we will use
//          is from 0-1200 by 0-1200.  Therefore, we have Grid_Size=1201.
//
// GRID_SIZE_PLOTTED is the size of the map in pixels to be plotted.
//          We will plot the entire grid so the size should be 1200.
//               i.e. plot 0-1200 x 0-1200.
//
// SIZE_OF_SITES is the radius in pixels of the size of the goals and
//          dangers to be plotted.
//
// MAX_HEIGHT is the maximum altitude of the grid.
//
//
//

int const Grid_Size=1201;
int const Grid_Size_Plotted=1200;
int const Size_of_Sites=5;
int const Max_Height=5000;

//
//
// MAX_DISTANCE is used in global.c for finding the closest point.  This
//          is the initial value given.  i.e. INT_MAX.
```

```
//
// MAX_SLOPE is used in line.c for a value for a horizontal line.
//
//

long const Max_Distance=(3000*3000) + (2*Grid_Size*Grid_Size);
int const Max_Slope=10*Grid_Size;


#endif
```

C.2. Coordinate.h

```
#ifndef COORDINATE
#define COORDINATE

#include "list.h"

enum ListType {GOALLIST, GOALLISTCOPY, PLANLIST, DANGERLIST};


//_____________________________________________________________________________
//
//
//
//      The Coordinate class contains the x,y, and z coordinates for use
//              throughout the entire program.
//
//
//      The function CLOSERCOORDINATE takes two coordinates.
//              They are two goal positions.  CLOSERCOORDINATE returns
//              the closer coordinate to the coordinate itself.
//          The minimum criterion is the DistanceSquared2D function.
//
//              Note: If the points are equidistant, the first coordinate
//                      is returned.
//
//      The function HOWFAR takes a coordinate and returns the square
//              of the distance from itself to the coordinate as defined
//              by the DistanceSquared2D function.
//
//
//
//
// Quick reference:
//
//          Coordinate x;                   Sets x,y,z to 0,0,0
//          Coordinate x(1,2,3);            Sets x,y,z to 1,2,3
//          Coordinate x(1,2,3,GOALLIST);   Sets x,y,z to 1,2,3 and adds
//                                          the coordinate to the
//                                          global GoalList
//          x.x_coordinate;                 Returns x's x coordinate
//          x.CloserCoordinate(A,B);        Returns point A or B, whichever
//                                          is closer to x.
//          x.HowFar(A);                    Returns the square of the
//                                          distance from x to A
//                                          using 2D
//          DistanceSquared3D(A,B);         Returns the square of the
```

```
//                                          distance from A to B
//          DistanceSquared2D(A,B);         Returns the square of the
//                                          distance from A to B
//                                          (x,y) only
//
//


class Coordinate
{
        public:
                Coordinate() {x_coordinate=y_coordinate=z_coordinate=0;}
                Coordinate (int, int, int);
                Coordinate ( Coordinate const &from )
                                        {if ( &from != this )
                                        {   *this = from; }}
                Coordinate (int, int, int, ListType );
                int x_coordinate;
                int y_coordinate;
                int z_coordinate;


                Coordinate* CloserCoordinate (Coordinate *, Coordinate *)
                                                        const;
                long HowFar (Coordinate *) const;
};

//_____________________________________________________________________
//
//
//
//      The operator << is overloaded to handle printing Coordinates.
//
//

ostream& operator<< (ostream& , Coordinate const &);


//
//
//_____________________________________________________________________
//
//
//
//      The inline function DISTANCESQUARED3D computes the square of the
//          distance between 2 Coordinates.  The reason actual distance
//          is not computed is because it is too expensive to compute the
//          square root.
//
//

inline long DistanceSquared3D (Coordinate const *First,
                               Coordinate const *Second)
{
        return ((
                        ((long)(First->x_coordinate-Second->x_coordinate)*
                         (First->x_coordinate-Second->x_coordinate))
                                        +
                        ((long)(First->y_coordinate-Second->y_coordinate)*
                         (First->y_coordinate-Second->y_coordinate))
                                        +
```

```
                    ((long)(First->z_coordinate-Second->z_coordinate)*
                     (First->z_coordinate-Second->z_coordinate))));
}


//
//_____________________________________________________________________________
//

//
//
//      The inline function DISTANCESQUARED2D computes the square of
//          the distance between 2 Coordinates (x,y only).
//          The reason actual distance is not computed is because
//          it is too expensive to compute the square root.
//
//

inline long DistanceSquared2D (Coordinate const *First,
                               Coordinate const *Second)
{
        return ((
                    ((long)(First->x_coordinate-Second->x_coordinate)*
                     (First->x_coordinate-Second->x_coordinate))
                                +
                    ((long)(First->y_coordinate-Second->y_coordinate)*
                     (First->y_coordinate-Second->y_coordinate))));
}

#endif
```

C.3. Global.h

```
#ifndef GLOBAL
#define GLOBAL
#include <stream.h>
#include "list.h"
#include "coordinate.h"
#include "constants.h"

//_____________________________________________________________________________
//
/*


                GLOBAL VARIABLES (and ENUMERATED TYPES)

*/
//_____________________________________________________________________________
//
//
//
// MAKELIST MACRO
//
//
// Create a list class that takes pointers to Coordinates.
//    MyList is the name of the List class.
//
```

```
//    ex) MyList x;  // declares x as a list structure that takes *Coordinate
//
//
//
// GOAL LIST
//
//    The GoalList is a list of points that the aircraft is to visit.
//          This list should not be altered.
//          Note: Check the List class to find all of the options
//                that can be performed on list objects.
//
//          ex) GoalList.next() returns a Coordinate *
//
//
//
// GOAL LIST COPY
//
//    The GoalListCopy is a copy of the GoalList that we can freely
//          add and delete from.
//
//
// DANGER LIST
//
//    The DangerList is a list of points that are believed to be a
//          potential danger and thus should be avoided.
//
//          This list should not be altered.
//
//
// STARTING POINT
//
//    The StartingPoint is simply a * Coordinate.
//
//
// ENDING POINT
//
//    The EndingPoint is simply a * Coordinate.
//
//
// PLAN LIST
//
//    The PlanList is a list of points corresponding to a path from the
//          Starting Point to the Ending Point via a navigational
//          plan.  This list is erased before any navigation is used.
//          However, multiple Plans are permitted for graphing.
//
//
// MAP[x][y]
//
//    The Map[][] is an array of points corresponding to the terrain.
//          The first coordinate is x and the second is y.
//
//    This is a Grid_Size x Grid_Size point grid.  See constants.h
//
//
// GRAIN
//
//    Grain is the grain of the map to be plotted.  A grain of 1 will plot
//          every point. (this is the default)
//          A grain of 5 will plot every 5th point (etc).
//
```

```
//
// GLOBALDANGERRADIUS
//
//    Global Danger Radius is the radius of the dangers that the
//          plane cannot enter.  This is the distance from the
//          danger (infinite cylinder context)
//
//
// HOWFARBACK
//
//    HowFarBack is an integer used with BestPrevious.  It is initially
//          set to 10 so that we look back 10 points.
//          In the restart operation in user.c, it is set to 1
//          so that any previous plan is ignored.
//
//
// PLANE_LENGTH_CONSTRAINT
//
//    Plane_Length_Constraint is the minimum distance that a plane
//          has to travel in a specific direction before it can
//          make a 45 degree turn.  The value is measured in pixels
//
//
// LINE_OF_SITE
//
//    Line_of_site is the maximum distance that the plane can "sense"
//          danger areas.  This is measured in pixels and it is a
//          radius.
//
//
// SHOWREGION
//
//    ShowRegion is a variable to toggle turning on the plotting danger
//          regions.  It defaults to yes (1).
//
//
//_____________________________________________________________________
//
/*

                GLOBAL FUNCTIONS (and INLINE FUNCTIONS)

*/
//_____________________________________________________________________
//
//
//
// FINDCLOSESTPOINT
//
//
//      The function FindClosestPoint simply takes a coordinate and uses
//              the DistanceSquared2D function to find the closest goal
//              in the List specified.
//
//
//    ex) FindClosestPoint (x,GOALLIST);  returns the point in the
//                                        GoalList that is the closest
//                                        to x. (including x if in list)
//
//                                        An error is retuned if the
//                                        list is empty.
```

```
//
// ABS
//
//
//    The function Abs takes a double and returns its absolute value
//
//    ex) double x=Abs(-9.32444);          // x=9.32444
//
//
// ROUND
//
//    The function Round takes a double and rounds it to the nearest
//          integer.
//
//    ex) int x=round(-9.32444);           // x=-9
//
//
// APPEND
//
//    The function Append takes a Coordinate * and a ListType.  The
//          Coordinate is added to the appropriate List.
//
//    ex) Append(NewPoint, DANGERLIST);    // puts NewPoint on the
//                                         // Danger list
//
//          Note: Coordinate *NewPoint = new Coordinate (*somecoord)
//          or    Coordinate *NewPoint = new Coordinate (1,2,3);
//
//
// ALTITUDE
//
//    The function Altitude takes an x and a y coordinate and finds
//          the correct z coordinate according to the MAP database.
//
//    ex) int z=Altitude(4,5);             // z = 2692
//
//
// BESTPREVIOUS
//
//    This function is used to get a previous point in the plane's
//          current path in order to determine the direction it
//          is headed.  Usually, we take the current point and
//          a point that was 10 steps previous in the plan
//          and calculate a slope from that.
//
//          The function looks into the PlanList and returns
//          a point that was 10 steps ago unless there are not
//          that many steps in the plan (at the beginning).
//
//_____________________________________________________________________
//




//
//
//  enum ListType {GOALLIST, GOALLISTCOPY, PLANLIST, DANGERLIST};
//
//                                        actually declared in
//                                        coordinate since it
```

```
//                                          needs to know first.

MakeList(Coordinate, MyList);
extern MyList GoalList;
extern MyList GoalListCopy;
extern MyList DangerList;
extern Coordinate *StartingPoint;
extern Coordinate *EndingPoint;
extern MyList PlanList;
extern short Map[Grid_Size][Grid_Size];
extern int Grain;
extern int GlobalDangerRadius;
extern int HowFarBack;
extern int Plane_Length_Constraint;
extern long Line_of_Site;
extern int ShowRegion;

Coordinate* FindClosestPoint (Coordinate *point, ListType a_plan);
double abs (double number);
int round (double number);
void Append (Coordinate *, ListType );
int Altitude (int , int );
Coordinate* BestPrevious();

#endif
```

C.4. Line.h

```
#ifndef LINE
#define LINE

#include "coordinate.h"
#include "global.h"
#include "potential.h"


//_____________________________________________________________________________
//
//
//
//
//      The class LINE handles finding a line between two Coordinates as
//          MIDPOINTLINE does.  Specifically, this function does not
//          do anything with the z_coordinate.  It simply copies the
//          correct altitude from the map directly.  The line is between
//          the x and y coordinates.  This method can accommodate
//          the use of potential fields by initializing the constructor
//          with a 1 ( x=new Line(1) ).  All other values will not
//          use a potential field.
//
//    The difference is that this method uses the equation of the line
//          to determine the points.
//
//    If |m|>1 we increment y and find x.
//    If |m|<=1 we increment x and find y.
//
//
//    MYFIELD is an instance of potential.  It is used when a potential
```

```
//          field for calculating the path is desired. (see potential.h)
//
//
//    GETEQUATION figures out the equation of the line from start
//          to finish.
//
//    GETFIELDS figures out the surrounding potential field values
//          around the point that was last put into the PLanList.
//
//    GETPOINT ultimately figures out an alternative point
//          (not necessarily in a straight line) because we would
//          otherwise exceed out Threshold value. (see potential.h)
//
//          We have a view defined as     0 1 2
//                                        7 X 3
//                                        6 5 4
//
//          First we figure out which direction we were going.
//          We try that direction and then direction+1 etc...
//          to get out.
//
//
//    GETOFFSET figures out which direction the plane was headed.
//          It uses this information to try to continue in this
//          direction.  It goes to it's right if it can't go the
//          way it was going.
//
//    SETPOSITION is used by GetPoint.  It sets a position based on
//          a clockwise motion looking for the lowest potential starting
//          at the same direction we were currently moving.
//
//
//    INITIALIZEPOTENTIAL is used by the Planner.  It is used before
//          we start finding a path with the potential field.
//
//
//    The FINDPATH method takes a starting point, a destination
//          point, and the name of the global list to add the path to.
//          It finds a path that is a straight a line as possible
//          using integer arithmetic.
//
//
//

class Line
{
      private:
            double slope;           // Slope of line m
            double constant;        // Constant of line b
            int Current_x;          // Our current x position
            int Current_y;          // Our current y position
            int New_x;              // used when deciding new direction
            int New_y;
            int potential;          // potential=1 => using a potential
                                    //                field
            int Field[8];           // used to find the surrounding
                                    //    field values in 8 directions

            Potential MyField;

            void GetEquation (int , int , int , int );
```

```
            void GetFields (int , int);
            void GetPoint (int);
            void SetPosition (int);
            int GetOffset();
        public:
            Line(int field) {potential=field; slope=0; constant=0;};
            void InitializePotential();
            void FindPath (Coordinate const*, Coordinate const*, ListType);
};

#endif
```

C.S. List.h

```
#ifndef LIST_H
#define LIST_H

//_____________________________________________________________________
//
//
//
//    The purpose of this class is to allow us to create a list that
//          can handle any type.  Specifically, MakeList(A,B) creates
//          a list that stores pointers to A in an object called B.
//
//    MakeList(A,B); B foo; B is a list class.  foo is the actual list.
//
//
//    The following is a summary of what you can do with a list:
//
//          GoalList.empty();       // returns an integer
//          GoalList.length();      // returns an integer
//
//          GoalList.resetMarker(Coordinate *); // all of these
//          GoalList.prepend(Coordinate *);     // return nothing
//          GoalList.append(Coordinate *);      //
//
//          GoalList.first();       // all return Coordinate *
//          GoalList.last();        //
//          GoalList.get();         //
//          GoalList.next();        //
//          GoalList.del(item);     // item must be in the list
//                                  //    and it is removed.
//                                  //    ( Coordinate *item )
//                                  //    returns nothing
//
//          cout << GoalList;
//
//_____________________________________________________________________
//


#include <stream.h>
#include <libc.h>
#include <generic.h>
```

```
struct link {
      void *data;
      link *next;
      link(void *in_data, link *in_next = 0) {data=in_data; next=in_next;}
};

class List {
      link *head, *tail, *mark;
      int len;

  public:
      List() {head=tail=0; len=0;}
      ~List();
      int empty() {return head==0;}
      int length() {return len;}
      void resetMarker() {mark=head;}
      void prepend(void *in_data);
      void append(void *in_data);
      void* first();
      void* last();
      void* get();
      void* next();
      void del(void *data);
      virtual ostream& printItem(ostream& out, void *data) const = 0;
};

ostream& operator<< (ostream &out, List &list);


#define MakeList(elt_type, name)                                          \
      class name : public List {                                          \
      public:                                                             \
            elt_type* first() {return (elt_type*)List::first();}          \
            elt_type* last() {return (elt_type*)List::last();}            \
            elt_type* get() {return (elt_type*)List::get();}              \
            elt_type* next() {return (elt_type*)List::next();}            \
            virtual ostream& printItem(ostream &out, void *data) const    \
                  { return out << *((elt_type const*)data); }             \
}


#endif
```

C.6. MidpointLine.h

```
#ifndef MIDPOINT_LINE
#define MIDPOINT_LINE
#include "coordinate.h"
#include "global.h"


//_____________________________________________________________________
//
//
//
//
//    The class MIDPOINTLINE handles finding a line between
```

```
//          two Coordinates.  Specifically, this function does not
//          do anything with the z_coordinate.  It simply looks up the
//          correct altitude.  The line is between the x and y coordinates.
//
//    The FINDPATH method takes a starting point, a destination
//          point, and the name of the global list to add the path to.
//          It finds a path that is a straight a line as possible
//          using integer arithmetic.  This method is called Scan Line
//          Conversion.
//
//          NOTE:
//                This method may not converge!!!
//
//                The reason is that the order must be smaller
//                      x to larger x and in a positive slope.
//
//

class MidpointLine
{
      public:
            MidpointLine() {};
            void FindPath(Coordinate const*, Coordinate const*, ListType);
};

//_____________________________________________________________________________
//


#endif
```

C.7. NewLine.h

```
#ifndef NEWLINE
#define NEWLINE

#include <iostream.h>
#include <ctype.h>
#include <stdlib.h>
#include <math.h>
#include "coordinate.h"
#include "global.h"
#include "turnaround.h"


//_____________________________________________________________________________
//
//
//
//
//    The class NEWLINE handles finding a line between two Coordinates as
//          LINE does. The difference is that this method is capable
//          of avoiding dangers.  With the constructor passed a 1,
//          The plane knows in advance where all of the dangers are.
//          Any other value passed to the constructor assumes
//          that the plane can only sense danger as far as the distance
//          LINE_OF_SITE is defined.
//
```

```
//
//    KNOWINADVANCE is the variable that distinguishes between the
//          model where the plane knows where all of the dangers
//          are ahead of time (KnowInAdvance=1) or it doesn't
//          (any other value).
//
//
//    DANGERRADIUS is the effective distance of the danger points.
//          The plane cannot fly this close to a danger point.
//          DangerRadius=OldRadius*OldRadius
//
//
//    OLDDANGERRADIUS is the effective radius of the danger points.
//          The plane cannot fly this close to a danger point.
//          This value is worth keeping around for later use.
//
//
//    SLOPE_TRAVEL is the slope of the current direction we are
//          traveling.
//
//
//    CONSTANT_TRAVEL is the constant "b" in the equation of the
//          line that we are traveling on.
//
//
//    CURRENTPOSITION_X is the current x position we are at.
//
//
//    CURRENTPOSITION_Y is the current y position we are at.
//
//
//    CLOSESTDANGER_X is the x position of the closest danger point.
//
//
//    CLOSESTDANGER_Y is the y position of the closest danger point.
//
//
//    CLOSESTSOFAR_X is the x position of the danger point that is the
//          closest so far to the plane.  This is used when the
//          plane can only sense the dangers in a certain radius.
//
//
//    CLOSESTSOFAR_Y is the y position of the danger point that is the
//          closest so far to the plane.  This is used when the
//          plane can only sense the dangers in a certain radius.
//
//
//    DANGERINTERCEPT_X is the x position of the intercept between
//          the current line of travel and a line perpendicular to
//          it through a danger point.  It is used to find the
//          distance between a point and a line.
//
//
//    DANGERINTERCEPT_Y is the y position of the intercept between
//          the current line of travel and a line perpendicular to
//          it through a danger point.  It is used to find the
//          distance between a point and a line.
//
//
//    SUBGOAL is 1 if there is a temporary subgoal to be visited
//          otherwise it is 0.
```

```
//
//
//    SUBGOAL_X is the x position of that subgoal if it exists.
//
//
//    SUBGOAL_Y is the y position of that subgoal if it exists.
//
//
//    DESTINATION is the original destination point to reach.
//
//
//    TURNCONSTRAINT is used to put a turning constraint on the
//          plane.
//
//
//    GETEQUATION figures out the equation of the line from start
//          to finish.
//
//
//    DISTANCEPOINTTOLINE calculates the distance of the point
//          given to it to the current line of travel.
//
//
//    RECORDCLOSESTDANGER finds the closest danger to the current
//          position and stores it in ClosestDanger_x and
//          ClosestDanger_y.
//
//
//    CHECKCLOSEST makes sure that the danger passed to it is in
//          the line of the path that it was given.  If there
//          is this path is a path to a subgoal, it allows
//          dangers to be slightly beyond the given path.
//
//
//    GETINTERMEDIATEPOINT calculates a subgoal if a danger is in
//          the way of the current path.
//
//
//    CHECKFORDANGERKNOW is used to determine if there is a danger
//          in the way and uses the other methods to get a
//          subgoal.  This method assumes the plane knows where
//          the dangers are.
//
//
//    CHECKFORDANGERDONTKNOW is used to determine if there is a danger
//          in the way and uses the other methods to get a
//          subgoal.  This method assumes the plane only knows where
//          the dangers are in a certain radius.
//
//
//    INRANGEFUNCTION returns a 1 if the danger is in range for
//          the method where the plane doesn't know in advance
//          where all of the dangers are.
//
//
//    INITIALIZE is used by the Planner.  It is used before
//          we start finding a path to initialize a danger radius.
//
//
//    SETDESTINATION is used by the Planner.  It is used before
//          we start finding a path to initialize a destination.
```

```
//
//
//    The FINDPATH method takes a starting point, a destination
//          point, and the name of the global list to add the path to.
//          It finds a path that is a straight a line as possible
//          using integer arithmetic.
//
//
//

class NewLine
{
      private:
            int KnowInAdvance;
            int DangerRadius;
            int OldDangerRadius;

            double slope_travel;          // Slope of line m
            double constant_travel;       // Constant of line b

            int CurrentPosition_x;
            int CurrentPosition_y;

            int ClosestDanger_x;
            int ClosestDanger_y;

            int ClosestSoFar_x;
            int ClosestSoFar_y;

            int DangerIntercept_x;
            int DangerIntercept_y;

            int subgoal;
            int subgoal_x;
            int subgoal_y;

            Coordinate *Destination;

            TurnAround TurnConstraint;

            void GetEquation (int , int , int , int );
            int DistancePointToLine (Coordinate *);
            int RecordClosestDanger ();
            int CheckClosest (int,int,int,int,int,int);
            Coordinate* GetIntermediatePoint ();
            void CheckForDangerKnow ();
            void CheckForDangerDontKnow ();
            int InRangeFunction();
      public:
            NewLine(int know) {     KnowInAdvance=know;
                                    slope_travel=0;
                                    constant_travel=0;
                                    subgoal=0;
                                    subgoal_x=-1;
                                    subgoal_y=-1;
                                    ClosestDanger_x=-1;
                                    ClosestDanger_y=-1;
                                    ClosestSoFar_x=-1;
                                    ClosestSoFar_y=-1;};
            void Initialize ();
            void SetDestination(int, int);
```

```
            void FindPath (Coordinate const*, Coordinate const*, ListType);
};

//_____________________________________________________________________________
//


#endif
```

C.8. Planner.h

```
#ifndef PLANNER
#define PLANNER
#include "global.h"
#include "turnaround.h"

//
// Include all of the point to point finders here.
//

#include "mdptline.h"
#include "y_equal_x.h"
#include "line.h"
#include "newline.h"


//_____________________________________________________________________________
//
//
//
//
//    The class PLANNER handles the navigation.  It uses the classes
//          that find a path from one point to another and constructs
//          a way to decide which goal to visit next.
//
//    In other words the planner decides which two points to travel from
//          and passes them to a specific pathfinder class
//          (line, mdptline, y_equal_x...).
//
//    Using the planner class, one only needs to say:
//
//          ex)   Planner Navigator;
//                Navigator.GreedyMdPtLineApproach (start, finish)
//                (start and finish are Coordinate *)
//
//          and the Planner will construct the points from start to
//          finish, visiting all the goals in the GoalList, and put
//          the points into the PlanList.
//
//
//
//    GREEDYMDPTLINEAPPROACH uses the mid point line (Scan Line Conversion)
//          technique to go from point to point.  It uses a Greedy
//          approach to decide which goal to visit next. I.e.
//          What is the closest Goal from where I am now?
//          Note: Dangers are ignored.
//
//
```

```
//    GREEDYCOORDINATELINEAPPROACH uses the y_equal_x point to point
//          approach.  I.e. change one or both coordinates by +/- 1
//          until we are at the goal.  It also uses a Greedy approach
//          to decide which goal is closer.  This approach also
//          uses the turnaround.h to impose turning constraints
//          as the plan travels from goal to goal.
//          Note: Dangers are ignored.
//
//
//    GREEDYSTRAIGHTLINEAPPROACH uses the line technique from the Line
//          class to go from point to point.  It also uses a Greedy
//          approach to go from goal to goal.  This method also has
//          a turning constraint.
//          Note: Dangers are ignored.
//
//
//    GREEDYSTRAIGHTLINEPOTENTIALAPPROACH uses the line technique
//          from the Line class to go from point to point.
//          It also uses a Greedy approach to go from goal to goal.
//          A potential field is incorporated to accommodate Dangers.
//          This method also has a turning constraint from goal to
//          goal and in avoiding the dangers.
//
//
//    GREEDYCOORDINATELINEPOTENTIALAPPROACH uses the line technique
//          from the Line class to go from point to point.
//          It also uses a Greedy approach to go from goal to goal.
//          A potential field is incorporated to accommodate Dangers.
//          This method also has a turning constraint from goal to
//          goal and in avoiding the dangers.
//
//
//    GREEDYNEWLINEAPPROACH uses a special technique to avoid dangers.
//          Otherwise, it travels in a straight line.  This method
//          assumes that all of the dangers are known.
//          This method also has a turning constraint from goal to
//          goal and in avoiding the dangers.
//
//
//    GREEDYNEWLIMITEDLINEAPPROACH uses the same method as
//          GreedyNewLineApproach but the dangers are not known in
//          advance.  The method avoids dangers that are in its
//          range as defined by Line_of_Site.
//          This method also has a turning constraint from goal to
//          goal and in avoiding the dangers.
//
//

class Planner
{
      private:
            TurnAround TurnConstraint;

            MidpointLine MyMdptLine;
            Y_Equal_X *MyPotentialYXLine;
            Y_Equal_X *MyYXLine;
            Line *MyPotentialLine;
            Line *MyLine;
            NewLine *MyNewLine;
            NewLine *MyNewLimitedLine;
```

```
    public:
        Planner (){ MyLine=new Line(0);
                    MyYXLine=new Y_Equal_X(0);
                    MyPotentialLine=new Line(1);
                    MyPotentialYXLine=new Y_Equal_X(1);
                    MyNewLine=new NewLine(1);
                    MyNewLimitedLine=new NewLine(0);};

        void GreedyMdptLineApproach(Coordinate*, Coordinate*);
        void GreedyCoordinateLineApproach(Coordinate*, Coordinate*);
        void GreedyStraightLineApproach(Coordinate*, Coordinate*);
        void GreedyStraightLinePotentialApproach
                                    (Coordinate*, Coordinate*);
        void GreedyCoordinateLinePotentialApproach
                                    (Coordinate*, Coordinate*);
        void GreedyNewLineApproach (Coordinate*, Coordinate*);
        void GreedyNewLimitedLineApproach (Coordinate*, Coordinate*);
};

#endif
```

C.9. Plotter.h

```
#ifndef PLOTTER
#define PLOTTER

#include <gl/gl.h>      // Graphics
#include <gl/device.h>  // Graphics
#include "global.h"
#include "coordinate.h"


//_____________________________________________________________________
//
//
//
//    The Plotter is for the Silicon Graphics package.  It is used
//         by the User class to control plotting.
//
//
//    DEFINE COLORS defines some colors we will use for the map, goals,
//         dangers, and plans in RGB mode.
//
//    PLANCOLOR is an integer corresponding to a color for the PlanList.
//         Initially, PlanColor= .
//                          0=Green
//                          1=Cyan
//                          2=Magenta
//                          3=Yellow
//
//    PREVIOUS[3] is a 3D array that keeps track of the previous point
//         plotted in the PlanList.  When we plot the PlanList,
//         we need to know the previous point  we may draw a line
//         from the previous point to the current point.  It is in the
//         order x,y,z.
//
//    SETPLANCOLOR looks at the value of PlanColor and sets the color
```

```
//          appropriately.
//
//    CHANGEPLANCOLOR increments the integer PlanColor by 1.  It makes
//          sure that the number is reset to 0 when it is too high.
//
//    SETUP sets up the prefsize, winopen, perspective, lookat and
//          creates a background of Black and sets the color to
//          grey in order to plot the map.  It also creates a light
//          source, z-buffering, and other Silicon Graphics features.
//
//    PLOTDANGERORGOALLIST will plot either of these lists in a special
//          way.  The DangerList will be Red and the GoalList will
//          be Blue.  Also, it plots the point and a line from that
//          point to Size_of_Sites points away in the following directions:
//
//                      N,NW,NE,E,W,S,SW,SE
//
//          so that it can be seen.  (see constants.h)
//
//    PLOTPOINT is for use with the PlanList.  It takes a Coordinate *.
//          First, it sets the PlanColor with SetPlanColor.  Then,
//          it checks to see if we have a previous value.  If so,
//          we plot a line from the previous point to the given
//          coordinate.  After that, we reset Previous.
//
//    PLOTMAP plots the entire map (or a grid of size Grid_Size_Plotter).
//          The argument is Grid_Size_Plotter since that's
//          how many pixels there are in the grid.  It is plotted by
//          taking a point and drawing a polygon from itself to x+Grain,
//          itself to y+Grain, and x+Grain,y+Grain.  It is a 4-sided
//          polygon. (see constants.h)
//
//    PLOTLIST plots a list.
//
//          ex) PlotList(GOALLIST);        // plots the GoalList
//
//          Note: if we are plotting the PlanList, we accommodate
//                multiple plans by checking for the end of one
//                plan and then changing the color.
//
//
//
//    PLOTALL is the only function that can be used by the User.
//          It plots the map, plots the DangerList, plots the GoalList,
//          and plots the PlanList.  Then, it waits until the left
//          mouse button is pressed.  The size is determined by
//          Grid_Size_Plotted.  (see constants.h)
//
//

class Plotter
{
      private:
            int PlanColor;
            short previous[3];
            void SetPlanColor();
            void ChangePlanColor();
            void SetUp(int );
            void PlotDangerOrGoalList (ListType );
            void PlotPoint (Coordinate *);
            void PlotMap (int , int );
```

```
        void PlotList (ListType );
        void DangerAreaColor (int, int);

    public:
        Plotter () {    previous[0]=previous[1]=previous[2]=0;
                        PlanColor=0;};
        void PlotAll();

};

#endif
```

C.10. Potential.h

```
#ifndef POTENTIAL
#define POTENTIAL

#include <iostream.h>
#include <ctype.h>
#include <stdlib.h>
#include "global.h"

//_____________________________________________________________________
//
//
//
//
//    The POTENTIAL class is for storing values at every map point
//        that somehow "relate" to the plane what is a good place
//        and what is a bad place.
//
//
//    INITIALIZEFIELD asks the user which potential field method
//        to use and initializes the field appropriately
//        depending on which method is chosen. Currently,
//        only the Infinite Cylinder method is relevant.
//        The Every other point method was used as a test and
//        the Altitude as a potential field has too many
//        complications.  The methods are described below.
//
//    The following methods are used for defining a way to assign
//        potential values to a grid.
//
//    1) EVERYOTHER was used to test the use of a potential field.
//        It assigns a 1 to every other point and a 0 to the rest.
//
//    2) INFINITECYLINDER defines a circular region around a danger point.
//        It assigns a large value to all the points in a radius
//        around the danger in the x-y plane.  One must fly around
//        this field.
//
//    3) ALTITUDEFIELD was a concept to force the plan to fly
//        only at low altitudes.  There are several problems with
//        this simplified approach.
//
//
//    EVALUATORFUNCTION is used by InfiniteCylinder to determine the
//        distance to the closest danger.
```

```
//
//
//    POTENTIALFIELD[x][y] is an array of potentials at the grid points
//          x,y.  PotentialField[100][200] corresponds to the
//          potential field value at the point (100,200,z) where
//          z is the altitude and irrelevant.
//
//
//    POTENTIALTHRESHOLD is the largest value that the plane can
//          travel on.
//
//
//    LARGESTPOTENTIAL is the largest potential field in the entire
//          region.
//
//
//    DANGERRADIUS is the effective distance that the plane must
//          be in order to clear the danger.  This is used with the
//          Infinite Cylinder method.
//
//
//    RETURNPOTENTIALFIELD returns the value of the field at the specified
//          coordinates.
//
//
//    RETURNPOTENTIALTHRESHOLD returns the value of the PotentialThreshold
//
//
//    RETURNLARGESTPOTENTIAL returns the value of the LargestPotential
//
//
//    INCREASEPOTENTIAL simply gives the field value at the coordinates
//          specified a value equal to the largest potential value
//          so that we can avoid backtracking.
//
//
//    INITIALIZE is a function that asks the user for a threshold value
//          and the value of the largest number that a field cell
//          will have.  It then calls InitializeField()
//
//


class Potential
{
      private:
            void InitializeField();

            void EveryOther();
            void InfiniteCylinder();
            void AltitudeField();
            long EvaluatorFunction(int , int);

            int PotentialField[Grid_Size][Grid_Size];
            int PotentialThreshold;
            int LargestPotential;
            long DangerRadius;            // for use with InfiniteCyl.
      public:
           Potential ();
```

```
            int ReturnPotentialField(int ,int );
            int ReturnPotentialThreshold();
            int ReturnLargestPotential();

            void IncreasePotential (int ,int );

            void Initialize();
};

#endif
```

C.11. Reader.h

```
#ifndef READER
#define READER

#include <iostream.h>
#include "global.h"
#include "coordinate.h"


//_____________________________________________________________________________
//
//
//
// This class is used by main.c to read in the Digital Terrain Data.
//
//    The data is read in from Map[0][0], Map[1][0], ... , Map[1200][0],
//                        Map[0][1], Map[1][1], ...
//                              ...
//                        Map[1200][0], ... , Map[1200][1200].
//
//
//    READBLOCK (size) was used to read in the data in it's old format
//          (not digital).  The Size is always 1200.  The Block reads in
//          1201 numbers.
//
//
//    READMAP read in the map in its old format.
//
//
//    WRITEMAPTODIGITAL writes the old map to a digital file.
//
//
//    READDIGITALMAP reads in the map in its current format.  The
//          file name is assumed to be "digitaldata."  After it is
//          read in, a check is made to see if the first point matches
//          to what it was supposed to be.  If it doesn't match, an error
//          appears.
//
//                Note: For ReadDigitalMap, the size of the file
//                      is automatically taken in to account.
//

class Reader
{
      private:
            int y_index;
```

```
            void ReadBlock (int );

      public:
            Reader () {y_index=0;};
            void ReadMap ();
            void WriteMapToDigital ();
            void ReadDigitalMap ();

 };

#endif
```

C.12. Turnaround.h

```
#ifndef TURNAROUND
#define TURNAROUND

#include "global.h"
#include "coordinate.h"


//_____________________________________ ____________________________________
//
//
//
//
//    The class TURNAROUND handles the turning constraint that we will
//          put on the plane.  It will add the appropriate turn
//          on the plan list.
//
//
//
//    LENGTH is = to Plane_Length_Constraint which defines the minimum length
//          that the plane must travel before it can change directions
//
//
//    DIRECTION represents a direction in the following way:
//
//                0 1 2
//                7 X 3
//                6 5 4
//
//          is a grid.  The "X" is the present location and the
//          eight directions represent NW, N, NE, E, SE, S, SW and W
//          respectively.
//
//
//    GONCRTH takes a starting point and an ending point and travels the
//          Length distance in that direction adding it to the PlanList
//          and returning a new current position.
//
//
//    GONORTHEAST ... GOWEST do the same exact thing as GoNorth does except
//          they each travel in their respective directions.
//
//
//    GODIRECTION takes a starting position and a direction (int) and
//          translates Direction into GoNorth, GoSouth, etc... and returns
```

```
//          the new present position.  The direction given to it, is the
//          direction which it should turn (right or left).
//
//
//    CALCULATEDIRECTION takes a starting point and an ending point
//          and figures out the direction that it is currently traveling and
//          and the direction it would like to end up traveling.  It
//          calculates this information by using the slope.
//
//                m>=2 or m<=-2 is North or South
//                m<=0.5 and m>=-0.5 is East or West
//                m>0.5 and m<2 is NorthEast or SouthWest  and
//                m<-0.5 and m>-2 is SouthEast or NorthWest
//
//
//    CHECKLEGAL takes two directions and returns 1 if they are the same
//          direction.  Otherwise it returns 0.  At first, we thought
//          a difference of one to be allowed but ultimately we desire a
//          strict 45 degree turn.
//
//
//    ADDSUBTRACT takes two directions and decides whether to go the
//          planes' left or right.  It chooses based on which direction
//          involves the least number of turns.  1=add (right) and 0=subtract
//          (left) If the turn is 180 degrees, it turns to the right by
//          default.
//
//
//    TURN takes a previous point, a current point, and a destination point.
//          It calculates the current direction and the desired direction.
//          It also executes the turn and returns the new current position.
//
//


class TurnAround
{
      private:
            int Length;
            int Direction;

            Coordinate* GoNorth (Coordinate *);
            Coordinate* GoNorthEast (Coordinate *);
            Coordinate* GoNorthWest (Coordinate *);
            Coordinate* GoSouth (Coordinate *);
            Coordinate* GoSouthEast (Coordinate *);
            Coordinate* GoSouthWest (Coordinate *);
            Coordinate* GoEast (Coordinate *);
            Coordinate* GoWest (Coordinate *);

            Coordinate* GoDirection (Coordinate *,int );

            int CalculateDirection (int ,int ,int , int);
            int CheckLegal (int , int);
            int AddSubtract (int , int);
      public:
            TurnAround () {};

            Coordinate* Turn (Coordinate *, Coordinate *, Coordinate *);

};
```

```
#endif
```

C.13. User.h

```
#ifndef USER
#define USER

#include <iostream.h>
#include <ctype.h>
#include <stdlib.h>
#include "coordinate.h"
#include "global.h"
#include "planner.h"
#include "reader.h"
#include "plotter.h"


//_____________________________________________________________________________
//
//
//
//    The User class is something for the main program to use.  It is
//          designed to create a user friendly environment.  There are
//          two functions for the main program to use: Chooser and
//          ReadInMap.
//
//
//    READINMAP simply reads in the digital data file as define by the
//          reader class.  It checks to make sure that Map[0][0], the
//          first point read in is 2746.  Also, it is assumed that the
//          filename is digitaldata.
//
//
//    CHOOSER is an infinite loop asking the user what they would
//          like to do.  The choices include:
//
//          0) quit
//          1) Read in Starting Point.
//          2) Read in Ending Point.
//          3) Read in a new goal list.
//          4) Read in a new danger list.
//          5) Plot Everything.
//          6) Set the Grain of the map.
//          7) Turn region shading on/off.
//          8) Print the StartingPoint.
//          9) Print the EndingPoint.
//          10) Print the Goals.
//          11) Print the Dangers.
//          12) Print the planned route (Plan List).
//          13) Set Turning Length constraint.
//          14) Set Line of Site constraint.
//          15) Greedy Straight Line Approach.
//          16) Greedy MidPoint Line Approach.
//          17) Greedy Coordinate Line Approach.
//          18) Greedy Straight Line with Potential Field Approach.
//          19) Greedy Coordinate Line with Potential Field Approach.
//          20) Greedy Straight Line avoid dangers know.
```

```
//          21) Greedy Straight Line avoid dangers don't know.
//
//    The Choices are designed to be easy to use.
//
//    When Reading in a point, if the altitude isn't known, you can enter
//          0 for an automatic lookup of the altitude.
//
//    When reading in a new goal/danger list, the filename is asked for.
//          Note: it is assumed that the file looks like
//
//          ex)   1 2
//                3 4
//                9 10
//                ....
//                      where the first number is the x coordinate,
//                            and the second number is the y
//                            coordinate.
//
//    The only concern is that one must remember to do things in this order:
//                1) All List Operations
//                2) All Plane constraint Operations.
//                3) All Navigation Operations.
//                4) Graphics
//
//-------------------------------------------------------------------------
//
//
//    EMPTYTHELIST is a method that given a ListType, it empties that
//          list.
//
//
//    SETGRAIN sets the grain of the map to be plotted.  It's default
//          is 1.  In other words, it plots every point.  A grain
//          of 5 would plot every 5th point.
//
//
//    SETPLANE LENGTH sets the minimum length the plane must travel
//          before it can turn 45 degrees.
//
//
//    SETLINEOFSITE sets the maximum distance that the plane can sense
//          a danger.
//
//
//    RESTART copies the current GoalList into
//          the GoalListCopy.  This is something you want to do
//          after any navigational plan and when reading in a new goal
//          list.
//
//
//    READINSTART reads in 3 coordinates for the Starting Point.  If the
//          3rd coordinate is unknown, it can automatically look up
//          the correct altitude when a 0 is entered.  If any of the
//          numbers aren't entered as integers, nothing changes.
//
//
//    READINEND is analogous to ReadInStart, except it is for the
//          EndingPoint.
//
//
```

```
//    READINNEWGOALS will read in a file of new goals for the GoalList.
//          This method automatically copies the new GoalList into
//          GoalListCopy.  It also empties both lists before adding.
//          It can be aborted by entering a nonexistent filename.
//
//                Note: you cannot have a goal or a danger
//                      closer than Size_of_Sites+1 pixels from
//                      any boarder.
//
//    READINNEWDANGERS is analogous to ReadInNewGoals except it is for
//          dangers.  There is no copy of this list.
//
//
//    READINOLDMAP reads in the map in the old format.
//
//



class User
{
      private:
            Planner Navigator;
            Reader Scan;
            Plotter Plot;

            void EmptyTheList (ListType);
            void SetGrain();
            void SetPlaneLength();
            void SetLineOfSite();

            void Restart();
            void ReadInStart();
            void ReadInEnd();
            void ReadInNewGoals();
            void ReadInNewDangers();

      public:
            User() {};
            void Chooser();
            void ReadInMap();
            void ReadInOldMap();
};

#endif
```

C.14. Y_Equal_X.h

```
#ifndef Y_EQUAL_X
#define Y_EQUAL_X

#include "global.h"
#include "coordinate.h"
#include "potential.h"



//_____________________________________________________________________
```

```
//
//
//
//    Y_Equal_X is a method to go from one point to another.  The way
//          this method accomplishes that is at each step, if the
//          current position is not at the goal position, one or
//          both x and y coordinates are changed by 1 to get there.
//
//    ex)   1,2 to 5,7
//          1,2 - 2,3,- 3,4 - 4,5 - 5,6 - 5,7
//
//          We can also accommodate a potential field by creating
//                Y_Equal_X Mine = new Y_Equal_X (1);
//
//
//    The methods in this class that deal with potential fields
//    are analogous to the Line class.
//
//
//    CURRENT_X and CURRENT_Y is the current position that we are at.
//
//
//    NEW_X and NEW_Y is the position used for deciding the new direction
//
//
//    POTENTIAL if potential=1 then we are using a potential field
//          otherwise we are not.
//
//
//    FIELD[] is used to find the surrounding potential fields in 8
//          directions
//
//
//    MYFIELD is an instance of potential.  It is used when a potential
//          field for calculating the path is desired. (see potential.h)
//
//
//    GETEQUATION figures out the equation of the line from start
//          to finish.
//
//
//    GETFIELDS figures out the surrounding potential field values
//          around the point that was last put into the PLanList.
//
//
//    GETPOINT ultimately figures out an alternative point
//          (not necessarily in a straight line) because we would
//          otherwise exceed out Threshold value. (see potential.h)
//
//          We have a view defined as      0 1 2
//                                         7 X 3
//                                         6 5 4
//
//          First we figure out which direction we were g
//                We try that direction and then direction  ....
//                to get out.
//
//
//    GETOFFSET figures out which direction the plane was headed.
//          It uses this information to try to continue in this
//          direction.  It goes to it's right if it can't go the
```

```
//          way it was going.
//
//
//    SETPOSITION is used by GetPoint.  It sets a position based on
//          a clockwise motion looking for the lowest potential starting
//          at the same direction we were currently moving.
//
//
//    INITIALIZEPOTENTIAL is used by the Planner.  It is used before
//          we start finding a path with the potential field.
//
//
//    The FINDPATH method takes a starting point, a destination
//          point, and the name of the global list to add the path to.
//          It uses the method described.
//
//

class Y_Equal_X
{
      private:
            int Current_x;
            int Current_y;
            int New_x;
            int New_y;
            int potential;
            int Field[8];

            Potential *MyField;

            void GetFields (int, int);
            void GetPoint (int);
            void SetPosition (int);
            int GetOffset ();
      public:
            Y_Equal_X(int field) {  potential=field;
                                    MyField=new Potential();};
            void InitializePotential();
            void FindPath(Coordinate const *, Coordinate const *, ListType);

};


#endif
```


U.S. GOVERNMENT PRINTING OFFICE 750-113